

# Программирование Python

## Лекция 11

Кафедра ИВТ и ПМ  
ЗабГУ

2018

# План

## Прошлые темы

## Функции

- Глобальные и локальные переменные

- Параметры функций

- Возвращаемые значения

- Передача параметров по ссылке и по значению

- Чистые функции

# Outline

## Прошлые темы

### Функции

Глобальные и локальные переменные

Параметры функций

Возвращаемые значения

Передача параметров по ссылке и по значению

Чистые функции

- ▶ Что такое стандарт оформления кода?

- ▶ Что такое стандарт оформления кода?
- ▶ Как формулируется теорема о структурном программировании?

# Отладка

- ▶ Что такое отладка?
- ▶ Какие ошибки позволяет обнаружить отладка?
- ▶ Что такое трассировка?
- ▶ Что такое точка останова?

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

Параметры функций

Возвращаемые значения

Передача параметров по ссылке и по значению

Чистые функции

**Подпрограмма** (subroutine) — поименованная или иным образом идентифицированная часть компьютерной программы.

В Python подпрограммы называются **функциями**.



# Функции

Подобно переменным, функции в Python нужно определять перед использованием.

Определение функции

```
def имя_функции( параметр1, параметр2, ... ):
    global глобальные переменные # не обязательно
    оператор1
    оператор2
    ...
    return значение # не обязательно
```

# Функции

```
def имя_функции( параметр1, параметр2, ... ):
    global глобальные переменные # не обязательно
    операторы
    ...
    return значение # не обязательно
```

**def** - оператор объявления функции

**global** определяет список глобальных (внешних) переменных, которые будут использованы функцией. Приводятся только пользовательские переменные.

**return** - завершает подпрограмму и возвращает значение.

# Функции

Самая простая функция:

```
def foo():  
    pass
```

- ▶ не принимает параметров
- ▶ ничего не делает

# Функции

```
def foo():  
    print("Задача № 123")  
    print("Автор: Иванов. А. Б.")
```

Тело подпрограммы заканчивается там, где заканчивается вложенный блок.

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

Параметры функций

Возвращаемые значения

Передача параметров по ссылке и по значению

Чистые функции

# Глобальные и локальные переменные

**область видимости (scope)** - область программы, в пределах которой идентификатор (имя) некоторой переменной продолжает быть связанным с этой переменной и возвращать её значение

# Глобальные и локальные переменные

**область видимости (scope)** - область программы, в пределах которой идентификатор (имя) некоторой переменной продолжает быть связанным с этой переменной и возвращать её значение

**глобальной переменной** называют переменную, областью видимости которой является вся программа

**локальной переменной** называют переменную, объявленную внутри блока кода, например внутри функции.

# Глобальные и локальные переменные

```
def foo():  
    x = 1  # локальная переменная  
    y = 2  # локальная переменная  
    print(z)  # z - глобальная переменная
```

```
z = 300  # глобальная переменная  
# x и y здесь не видны
```

```
foo()
```



## Проблема?

```
L = [1.0823, 2.2221, 3.872]
```

```
for e in L:  
    print("{:.2}".format(e), end="")
```

```
for i in range( len(L)):  
    L[i] = L[i]**2
```

```
for e in L:  
    print("{:.2}".format(e), end="")
```

## Проблема?

```
L = [1.0823, 2.2221, 3.872]
```

```
for e in L:  
    print("{:.2}".format(e), end="")
```

```
for i in range( len(L)):  
    L[i] = L[i]**2
```

```
for e in L:  
    print("{:.2}".format(e), end="")
```

Код для вывода списка вещественных чисел на экран повторяется.

## Решение

Представить повторяющийся код в виде функции, вызывать функцию:

```
# зарезервированное слово def позволяет объединить  
# вложенный код в функцию  
# этот код будет выполнен только тогда,  
# когда функция будет вызвана
```

```
def print_list():  
    for e in L:  
        print("{:.2}".format(e), end="")
```

```
L = [1.0823, 2.2221, 3.872]
```

```
print_list()
```

```
for i in range( len(L)):  
    L[i] = L[i]**2
```

```
print_list()
```

# Глобальные переменные

Переменная L - глобальная. Значит она доступна во всей программе.

Однако стоит помнить, что любая переменная "видна" только в коде приведённом *после* её объявления.

```
1 def print_list():
2     for e in L:
3         print("{:.2f}".format(e), end=" ")
4
5 print_list()
6 L = [1.0823, 2.2221, 3.872]
```

## Функции. Рекомендацию по стилю

- ▶ Функции располагаются в самом начале программы, после подключения модулей и объявления глобальных переменных.
- ▶ Одну функцию от другой следует отделять двумя пустыми строками
- ▶ Имя функции должно быть лаконичным: отражать её назначение и быть кратким

# Функции и логическая структура программы

```
# Описание программы, автор
```

```
# подключение модулей
```

```
# глобальные переменные
```

```
# описание функций
```

```
def foo():
```

```
    . . .
```

```
def bar():
```

```
    . . .
```

```
# основная программа:
```

```
# ввод данных
```

```
# обработка
```

```
# вывод данных
```

# Глобальные переменные

Изменение глобальных переменных внутри функций.

Проблема:

```
def foo():  
    x = 42  # создание локальной переменной
```

```
x = 0  
foo()  
print(x)  # 0
```

В Python объявление переменное есть задание ей значения. Поэтому внутри функции foo объявлена новая (локальная) переменная, а не изменена глобальная.

**Локальные переменные при совпадении имён всегда "затеняют" глобальные.**

# Глобальные переменные

Изменение глобальных переменных внутри функций.

Решение:

```
def foo():  
    global x  
    x = 42  # изменение глобальной переменной  
  
x = 0  
foo()  
print(x)  # 42
```

Чтобы изменить внутри функции глобальную переменную, нужно внутри этой функции указать после служебного слова `global` имя переменной.



# Глобальные переменные

Чтобы изменить несколько глобальных переменных внутри функции, нужно их перечислить после `global`

Решение:

```
def foo():  
    global x, y, s  
    x = 42  
    y = 43  
    s = "abc"
```

```
x,y = 0,0  
s = "qwerty"  
foo()  
print(x)  # 42  
print(y)  # 43  
print(s)  # "abc"
```

## О использовании глобальных переменных

Глобальная переменная - один из путей передачи "внешних" данных в функцию.

Недостатки глобальных переменных

- ▶ Такой подход часто лишает функцию гибкости - глобальная переменная жёстко "зашита" в теле функции. Но что если нужно выполнить всё работу, которую выполняет функция, с другой глобальной переменной?

При вызове функции непонятно какие глобальные переменные она использует. Приходится изучать исходный код каждой функции чтобы это выяснить.

**Поэтому использование глобальных переменных следует свести к минимуму, отказываясь от них когда это возможно.**

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

**Параметры функций**

Возвращаемые значения

Передача параметров по ссылке и по значению

Чистые функции

# Параметры функций

Проблема:

```
def print_list():  
    for e in L:  
        print("{:5.2}".format(e), end="")  
    print()
```

```
L = [1.0823, 2.2221, 3.872]
```

```
print_list()
```

```
L2 = []
```

```
for i in range( len(L)):  
    L2 += [ L[i]**2 ]
```

```
print_list()  # как напечатать L2 ?
```

# Параметры функций

Решение проблемы глобальных переменных - организовать новый способ передачи информации в функцию.

Параметры при объявлении функции указываются в круглых скобках через запятую:

*# функция с двумя параметрами*

```
def foo(x, y):  
    print(x+y)
```

Такие параметры называются формальными.

**Формальный параметр** - аргумент, указываемый при объявлении или определении функции.

# Параметры функций

```
# функция с двумя параметрами  
# x, y - формальные параметры  
def foo(x, y):  
    print(x+y)
```

При вызове функции также также нужно будет указать её аргументы - фактические параметры.

```
...  
# 20, 30 - фактические параметры  
foo( 20, 30)  # 50
```

**Фактический параметр** — аргумент, передаваемый в функцию при её вызове

# Параметры функций

Формальные параметры можно рассматривать как локальные переменные внутри функции\*

Фактические - как значения этих формальных переменных\*\*

Имена формальных переменных от вызова к вызову функции не изменяются.

Значения фактических переменных могут меняться.

# Параметры функций

```
# x, y - формальные параметры  
def foo(x, y):  
    print(x+y)
```

```
foo(1, 2)  # 3
```

```
# на месте фактических параметров  
# могут быть глобальные переменные  
a = 7  
foo(a, 5)  # 12
```

```
# на месте фактических параметров могут быть выражения  
foo(2*4, 5)  # 13  
foo(1, a**2 + 1)  # 51
```



# Параметры функций

Как решить проблему?

```
def print_list():  
    for e in L:  
        print("{:5.2}".format(e), end="")  
    print()
```

```
L = [1.0823, 2.2221, 3.872]
```

```
print_list()
```

```
L2 = []  
for i in range( len(L)):  
    L2 += [ L[i]**2 ]
```

```
print_list()  # как напечатать L2 ?
```

## Параметры функций

Объявить формальный параметр в функции, через который передавать список:

```
def print_list(l):  
    for e in l:  
        print("{:5.2}".format(e), end="")  
    print()
```

```
L = [1.0823, 2.2221, 3.872]
```

```
print_list(L)
```

```
L2 = []  
for i in range( len(L)):  
    L2 += [ L[i]**2 ]
```

```
print_list(L2)
```

# Параметры функций

Контролировать тип фактических переменных - задача программиста:

```
def foo(x, y):  
    print(x+y)
```

```
foo(10, "123")
```

```
Traceback (most recent call last):
```

```
File "my_progr.py", line 5, in <module>
```

```
    foo(10, "123")
```

```
File "my_progr.py", line 2, in foo
```

```
    print(x + y)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Документирование функций

Чтобы у программиста использующего функцию не возникало проблем с определением типа и смысла параметров, нужно указывать их в комментарии.

Это особенно важно для сложных функций, в которых трудно сразу понять тип и смысл параметров

Хорошим тоном считается также пояснение назначения функции

```
def foo(x, y):  
    # выводит на экран сумму аргументов x, y  
    # x, y - целый или вещественный тип  
    print(x+y)
```

# Параметры функций

При создании функции нужно задавать вопрос: "Какие параметры она должна иметь?"

С этой точки зрения хорошей можно назвать ту функцию, которую не нужно переделывать если изменятся используемые ей данные.

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

Параметры функций

**Возвращаемые значения**

Передача параметров по ссылке и по значению

Чистые функции

# Возвращаемые значения

Проблема:

```
def foo(x,y,z):  
    z = ( x + y )**2  
  
z = 0  
foo(1,2,z)  
  
print(z)    # 0  
# z не изменилась
```

Как получить данные из функции?

## Возвращаемые значения

Один из способов получения данных из функции - возвращаемые значения.

Оператор **return** используется чтобы вернуть значения из функции.

Функцию возвращающую результат можно использовать в качестве правого операнда оператора присваивания.

```
def foo(x,y):  
    return ( x + y )**2
```

```
z = foo(1,2)  
print(z)    # 9
```

```
z = foo(3,4)  
print(z)    # 25
```



# Возвращаемые значения

Функция может возвращать несколько значений - кортеж.

```
def foo(x,y):  
    return x+y, x*y
```

```
a,b = foo(1,2)  # 3, 2
```

```
a,b = foo(3,4)  # 7, 12
```

## Возвращаемые значения

Для улучшения читаемости программы иногда лучше записывать значения во временные локальные переменные, а уже потом возвращать:

*# плохая читаемость*

```
def foo(x,y):  
    return x+y**2 - 1 - sin(x), (x-y)**0.5 + 0.5*ln(y)
```

*# хорошая читаемость*

```
def foo(x,y):  
    r1 = x+y**2 - 1 - sin(x)  
    r2 = (x-y)**0.5 + 0.5*ln(y)  
    return r1, r2
```

## Возвращаемые значения

Если внутри функции не используется оператор return или он используется без значения, то возвращаемое значение такой функции - **None**

```
def foo(x,y):  
    print(x+y)
```

```
def bar(x,y):  
    print(x*y)  
    return
```

```
a = foo(1,1)  # a = None  
b = bar(1,2)  # b = None
```

# Возвращаемые значения

Оператор `return` можно использовать не только для возвращения значения, но и для безусловного выхода из функции.

Операторы следующие за **`return`** никогда не выполняются:

```
def foo(x,y):  
    print(x+y)  
    return  
    print("Это никто не увидит")  
    print("И это тоже")
```

## Возвращаемые значения

Оператор `return` можно использовать для того чтобы сообщить, что в функции что-то пошло не так.

```
def foo(x,y):  
    print(x+y)  
    if y==0:  
        return -1  
    print(x/y)
```

```
a = 0  
if foo(1, a) == -1:  
    print("Некорректные входные данные функции ... ")
```

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

Параметры функций

Возвращаемые значения

**Передача параметров по ссылке и по значению**

Чистые функции

# Передача параметров по ссылке и по значению

Существуют два способа передачи фактических параметров в функцию:

- ▶ по ссылке
- ▶ по значению

## Передача параметра по значению

Изменение формального параметра внутри функции не влияет на фактический.

Это происходит потому, что внутри функции создаётся копия фактического параметра.

```
# x - формальный параметр
```

```
def foo(x):
```

```
# изменится локальная копия фактического параметра
```

```
    x = 1000
```

```
a = 42
```

```
foo(a) # a - фактический параметр
```

```
print(a) # 42
```

```
foo(12345) # 12345 - фактический параметр
```

Передавать по значению можно как переменные так и выражения.



## Передача параметра по ссылке

Изменение формального параметра внутри функции приводит к изменению фактического.

Поэтому передавать по ссылке можно только фактические параметры-переменные.

В Python, в отличие от многих других языков, нет синтаксических средств для того, чтобы указать способ передачи параметра.

Это сделано потому, что лучший (и самый очевидный) передать выходные данные из функции - это вернуть их с помощью оператора `return`.

# Передача параметров по ссылке и по значению

Однако, некоторые типы данных по умолчанию передаются по ссылке.

**Неизменяемые** типы данных передаются по значению

- ▶ bool
- ▶ int
- ▶ float
- ▶ complex
- ▶ str
- ▶ tuple

# Передача параметров по ссылке и по значению

Изменяемые типы данных передаются по значению

- ▶ list
- ▶ dict
- ▶ set
- ▶ class

## Передача параметра по значению

```
# x - формальный параметр
def foo(x):
    # изменится локальная копия фактического параметра.
    x = 1000

# l - формальный параметр
def bar(l):
    l[0] = 1000

a = 42
foo(a)  # a - фактический параметр
print(a)  # 42

L = [42]
bar(L)  # L - фактический параметр
print(L)  # [1000]
```

## Передача параметра по значению и по ссылке

Объекты созданные внутри функции существуют до конца выполнения функции.

Даже если присваивать их параметру переданному по ссылке.

```
def foo(x):  
    x = x*2
```

```
a = 7  
foo(a)  
print(a)    # 7
```

```
a = [1,2]  
foo(a)  
print(a)    # [1,2]
```

# Передача параметра по значению и по ссылке

Фактический параметр переданный по ссылке может быть *изменён* внутри функции.

```
def foo(x):  
    x += x
```

```
a = 7  
foo(a)  
print(a)    # 7
```

```
a = [1,2]  
foo(a)  
print(a)    # [1,2,1,2]
```

# Outline

Прошлые темы

## Функции

Глобальные и локальные переменные

Параметры функций

Возвращаемые значения

Передача параметров по ссылке и по значению

Чистые функции

# Чистые функции

**Побочный эффект функции** — возможность в процессе выполнения своих вычислений: читать и модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывать их обработчики.

Если функция меняет входной параметр, то она имеет побочный эффект.

Если вызвать функцию с побочным эффектом дважды с одним и тем же набором значений входных аргументов, может случиться так, что в качестве результата будут возвращены разные значения.



# Чистые функции

*# функция с побочными эффектами*

```
def foo1(a,b):  
    global c  
    c = a + b
```

*# функция с побочными эффектами*

```
def foo2(a,b):  
    c = a + b  
    print(c)
```

*# функция без побочных эффектов*

```
def bar(a,b):  
    return a+b
```

# Чистые функции

**Недетерминированность функции** — возможность возвращения функцией разных значений несмотря на то, что ей передаются на вход одинаковые значения входных аргументов.

# Чистые функции

*# недетерминированная функция*

```
def foo(a,b):  
    return random()*(b-a)+a
```

*# детерминированная функция*

```
def bar(a,b):  
    return a+b
```

# Чистые функции

**чистая функция**, это функция, которая:

- ▶ является детерминированной;
- ▶ не обладает побочными эффектами.

# Чистые функции

Так как результат чистых функций всегда предсказуем, транслятор может оптимизировать их вызовы: анализировать код и запоминать результаты функций с часто повторяющимся набором значений. Далее вместо вызова функции будет сразу подставлен её результат.

Кроме этого использование чистых функций позволяет избежать многих ошибок потому, что все данные изменяются явно.

**Поэтому следует стремиться к написанию чистых функций когда это возможно.**

# Ссылки и литература

Ссылка на слайды

[github.com/VetrovSV/Programming](https://github.com/VetrovSV/Programming)